QL Assembly Language Mailing List

Issue 002

Norman Dunbar

PUBLISHED BY MEMYSELFEYE PUBLISHING ;-)

First printing, July 2015

This pdf document was created on *D:20160209102713Z.*

Contents

List of Tables

List of Figures

Listings

1. Preface

1.1 Feedback

Please send all feedback to `assembly@qdosmsq.dunbar-it.co.uk`. You may also send articles to this address, however, please note that anything sent to this email address may be used in a future issue of the eMagazine. Please mark your email clearly if you do not wish this to happen.

This eMagazine is created in LaTeXsource format, aka plain text with a few formatting commands thrown in for good measure, so I can cope with almost any format you might want to send me. As long as I can get plain text out of it, I can convert it to a suitable source format with reasonable ease.

I use a Linux system to generate this eMagazine so I can read most, if not all, Word or MS Office documents, Quill, Plain text, email etc formats. Text87 might be a problem though!

1.2 Subscribing to The Mailing List

This eMagazine is available by subscribing to the mailing list. You do this by sending your favourite browser to `http://qdosmsq.dunbar-it.co.uk/mailinglist` and clicking on the link “Subscribe to our Newsletters”.

On the next screen, you are invited to enter your email address *twice*, and your name. If you wish to receive emails from the mailing list in HTML format then tick the box that offers you that option. Click the Subscribe button.

An email will be sent to you with a link that you must click on to confirm your subscription. Once done, that is all you need to do. The rest is up to me!

1.3 Contacting The Mailing List

I'm rather hoping that this mailing list will not be a one-way affair, like QL Today appeared to be. I'm very open to suggestions, opinions, articles etc from my readers, otherwise how do I know what I'm doing is right or wrong?

I suspect George will continue to keep me correct on matters where I get stuff completely wrong, as before, and I know George did ask if the list would be contactable, so I've set up an email address for the list, so that you can make comments etc as you wish. The email address is:

`assembly@qdosmsq.dunbar-it.co.uk`

Any emails sent there will eventually find me. Please note, anything sent to that email address will be considered for publication, so I would appreciate your name at the very least if you intend to send something. If you do not wish your email to be considered for publication, please mark it clearly as such, thanks. I look forward to hearing from you all, from time to time.

If you do have an article to contribute, I'll happily accept it in almost any format - email, text, Word, Libre/Open Office odt, Quill, PC Quill, etc etc. Ideally, a LaTeXsource document is the best format, because I can simply include those directly, but I doubt I'll be getting many of those! But not to worry, if you have something, I'll hopefully manage to include it.

2. Merry Christmas

It's probably a little late to wish you a Happy Christmas but this year all my followers received a free download of a slightly updated eBook containing all the articles published in *QL Today* over the past 'n' years (longer than I care to think about!)

All the diagrams etc have been converted from ASCII art to use proper png formats created with the "graphviz" utility (`http://www.graphviz.org/`). There was quite a bit of work involved in getting the old text, diagrams and code converted to LaTeX(that's how they like it to be written!) but I think, so far, everyone thinks that a good job has been done. Even if I say so myself.

The eBook has been typeset using LaTeXwhich is a professional system, much loved and used in academia and science for thesis[1] and scientific papers. I even have a couple, probably more, technical manuals written and published using LaTeX, my favourite being *Compiler Design in C* by Allen Holub. Speaking of compiler design, there are some relevant bits coming up in this issue on this matter, but don't panic!

If you have not already downloaded your free copy - it's in pdf format only at the moment, then please go to `http://qdosmsq.dunbar-it.co.uk/downloads/QLToday/QL_Assembly.pdf` and help yourself. I am looking into converting the pdf to other formats as some of my readers would like a version for Kindle and other eReaders.

Elsewhere in this issue, you will find some observations by George Gwilt on this eBook and its contents. George also has comments and observations on the first issue of this eMagazine, they are coming up next...

Happy new year, may all my readers enjoy a prosperous 2015.

[1] What is the plural of thesis?

3. Comments on Issue 1

This chapter is dedicated to George Gwilt's feedback on some of the content of the first issue of the eMagazine.

3.1 Special Programs

As Norman says, Special Programs are signalled by an extra word, $4AFB, after the program's name. In such a program there follows a set of instructions ending with RTS. These instructions are called as a subroutine inside the keywords EX, EW and ET (EX..) before they get around to creating and activating the job. That is why the instructions are obeyed "in the context of SuperBASIC", as are all keywords.

[ND] *Now that small explanation makes it all clear, which is more than the official docs have been able to do! (For me, anyway.)*

A description of how to write such a subroutine is given in section 3.5 of Jochen Merz's QDOS Reference Manual.

An example of the use of a special program is to be found in the SMSQ/E source code, in the program `extras_exe_source_cct_asm`. This program aims to concatenate a set of files and write them to an output file. All these files, including the output one, are to be named as channels to the program when it is executed. EX.. would happily try to open these, putting their IDs on the stack. If the output file does not exist, an error would be signalled by EX... It is to avoid this that the special program is invoked.

Having dealt with pipes and the parameter string, EX.. will go through the list of channels appearing, separated by commas, after the program name. But, just before this is done the special routine is called. This allows the program `..._cct_asm` to process the channels itself. It is thus able to open the output file as a new one, as well as processing the set of files to be concatenated. When it has finished doing this it amends the pointers to the parameters, set in A3 and A5, so that EX.. find that there are no channels to deal with before activating the job.

3.2 The Extra 12 bytes

The program in SMSQ/E which contains the code for EX.. is `sbsext_ext_exsbas_asm` which also contains the code for Special Programs. The program calculates how much data space to set for the Trap #1 call to MT_CJOB, which creates the job. If r is the number of channels and p the length of the parameter list, the addition made to the program's data space is:

$$4(r+3)+p$$

rounded up to even.

The reasoning is this. For each channel we need 4 bytes; for the parameter string we need p bytes, rounded up; for the two counts we need 4 bytes; and, just in case there are two pipes, we need a further 8 bytes. If both r and p are zero you can see that indeed 12 bytes are added. Also, if there are no pipes then 8 bytes have been added unnecessarily.

3.3 The EX Files - Page 6

When a job is activated the registers A4 to A7 are set as Norman describes apart from two details.

- The size of data space is $A5-A4$ not A5.
- A6 certainly points to the end of the internal job header, but not necessarily to the start of code. This is because, when a job is created by MT_CJOB, A1 either points to the absolute address of the start of code or is zero. Only when A1 is zero does A6 point to the start of code, which in this case does follow the internal header.

[ND] *Ugh! A silly mistake to make in the case of A5. I had also forgotten that a job can be created with the code immediately following the job header or with just a pointer to existing code elsewhere in memory.*

The facility to choose the address of the start of code allows the setting up of several programs, which have to be re-entrant, each with the same copy of code in ram, but, of course, with different data spaces and internal job headers

[ND] *A perfect example of this is Adrian Dickens' Self-cloning Program in chapter 4.4.2 of The QL Advanced User Guide, page 55. Unfortunately, as written, it uses an absolute address to fetch the SV_RAND word from the system variables, so probably will not work on many modern machines.*

3.4 LibGen Lite Errors

The program as it stands will not run on an unexpanded QL, though it should work on QPC2, for example. The reason is that 'buffer' is set up at an odd boundary. Oddly enough this is quite obvious from the example of the use of the program. This contains the line:

```
BUFFER EQU *+$000000017
```

This is an error I found myself committing again and again. It is caused by having an odd number of characters in the preceding string. Now, I always set strings using code which ends with:

```
DS.W 0
```

This sets the PC to the next even boundary from the end of the string. QPC2 has the 680020+ instruction set which allows word and long word accesses to an odd boundary which the 68000/8 does not.

[ND] *Another silly mistake. I could have sworn I used DS.W to ensure that storage was reserved and was on a word boundary, it turns out I used DS.B which gives rise to the problem George has pointed out above.*

3.5 ET Phone Home

I have never been able to use JMON. On the other hand I use QMON regularly. Indeed I did use it to go through the (working) version of LibGen_Lite I typed up. However, to call QMON I had to put a commma between QMON#6 and 23.

I should explain that I always use QMON in a daughter basic set up with #6 opened to a CON channel with window 512,204,0,0 and name 'x', for easy access. Also I had more programs running than Norman when I ET'd LibGen_Lite.

I'm afraid I used NET_PEEK to find LibGen_Lite's program number after ETing it. I also sometimes used NET_PEEK instead of QMON to see what was on the stack after ETing.

3.6 Actual Use of LibGen_Lite

Norman's program helps to solve a real problem. This is when you want to include an assembled program using the command LIB, which brings in the binary.

This has no symbols, so, if you want to access the program at some intermediate points you need to set up an appropriate set of equates.

This of course is what LibGen_Lite does. It will arrange for an appropriate SYM_LST file to be added just before the LIB which brings in the program itself.

A real example is the program PEAS_BIN, which is part of EasyPEasy. This program, will, on assembly, give rise to a SYM_LST file with over 50 entries. In fact only six of these are required to access the set of subroutines.

It would not matter too much if all the entries were included in the SYM_LST file, were it not that many of the unwanted equates referred to nondescript labels such as "l1" and "l2" which could well be used in the main program thus causing an error in assembly.

In fact I did find just that which made it impossible to use the complete SYM_LST file. Hence the abbreviated SYM_LST file published with PEAS_BIN.

So, the question is, how might one amend LibGen_Lite to produce a required subset of equates?

[ND] *One suggestion that comes rather quickly to mind, is to have a well documented format for the _SYM file which would allow the program to simply scan through looking for the "record type" that determines when a symbol is an offset to some code, or a simple EQU.*

I did originally write a small SuperBasic routine to extract the code offsets symbols and their values from a raw _SYM file, but was advised by George that the _SYM files created by GWASL and GWASS are different and that it would be best to use the textual SYM_LST files instead.

Another method would be for me to iron the bugs out of the full sized LibGen and get it working!

4. Chips and PEAs

George has some comments elsewhere in this issue on the Christmas eBook in which he mentions my lack of useful uses of the LINK, UNLK and PEA instructions.

I mentioned compilers earlier myself, so now is the time to combine these into an example of the use of these instructions, which were, apparently, originally designed for compiler writers to use. They are certainly useful for converting C code, for example, into assembler. Take the following *slightly* contrived C code:

```c
void addTwoNumbers(short value_a, short value_b, long *result)
{
    long temp;

    temp = value_a;
    temp += value_b;

    *result = temp;
}

int main(int argc, char *argv[])
{
    long Answer = 0;
    short a = 27;
    short b = 33;
    addTwoNumbers(a, b, &Answer);
    printf("a=%d, b=%d, a+b=%ld", a, b, Answer);
}
```

Listing 4.1: Contrived C Code

You can see that *addTwoNumbers* is a rather simple function that takes two values to be added together, *value_a* and *value_b*, which are short integers, or 16 bit words in assembly speak, and a pointer to a long integer, *result*. *Result* is the *address* of a 32 bit long word, where the calculated value will be stored.

The two numbers to be added are passed *by value*, while the result variable is passed *by reference*. This means that regardless of what the function does to the variables *value_a* and *value_b*, nothing will happen to them in the calling program as within the function, they are *copies* of the variables rather than the actual variables themselves.

In C, if you wish to amend a variable passed as a parameter to a function, then you must pass the address of the variable - a reference to the variable in other words. Mind you, the reference is itself passed by value as a copy of the real address, so the function cannot change the address, only what it points to.

Did I mention how simple compiler writing is? (George might have some comments to make here, he is Turbo Man these days and maintains and improves the much loved Turbo Compiler and toolkit.)

Yes, I know the C code above can be rewritten to be much much simpler, and to return the actual result through the normal manner in C, using the `return` command, but bear with me, I'm trying to demonstrate the `LINK`, `UNLK` and `PEA` instructions!

The local variable *temp* will have space allocated on the stack for it, and when the function ends, this temporary work space will be removed. With a compiler producing code for the Motorola 68000 series of processors, the code generated *could* resemble the following.

```

main            equ *

Answer          dc.l 0
a               dc.w 27
b               dc.w 33

                lea a,a0               Get address of a.
                move.w (a0),-(a7)      Stack a's value 27.
                lea b,a0               Get address of b.
                move.w (a0),-(a7)      Stack b's value 33.
                pea Answer             Stack reference to Answer.
                bsr addTwoNumbers
Stack_tidy      adda.l #2+2+4,a7       Tidy the stack.
                ...
```

Listing 4.2: Contrived Assembly Code

The code for main starts with some space allocated for the variables defined within the C code. Then, a copy of the values of variables *a* and *b* are pushed onto the stack, followed by the address of the variable *Answer*.

After the function call, these 8 bytes are tidied off the stack, before main carries on with whatever comes next.

The `PEA` instruction is roughly equivalent to the following code:

```
                lea Answer,a0
                move.l a0,-(a7)
```

Listing 4.3: PEA Equivalent Code

Back to the contrived example, the assembly code created for the function, might be as follows:

```
addTwoNumbers   equ *
```

```
        link     a6,-4              Local variable temp.
        move.l   a0,-(a7)           Save working register.
        clr.l    -4(a6)             Locals default to zero.
        move.w   $0e(a6),-4(a6)     Get value_a.
        add.w    $0c(a6),-4(a6)     Add value_b.
        move.l   $08(a6),a0         Fetch address of Answer.
        move.l   -4(a6),(a0)        Copy temp into Answer.
        move.l   (a7)+,a0           Restore working register.
        unlk     a6                 Clean up temp, a6 and a7.
        rts
```

Listing 4.4: Contrived Assembly Code - AddTwoNumbers

This code starts by creating space, 4 bytes, for the local variable named *temp* using the `LINK` instruction which creates a stack frame big enough to hold all the local variables required, and sets A6 to be the frame pointer. It does this *effectively* as per the following code:

```
        move.l   a6,-(a7)           Save current a6.
        movea.l  a7,a6              A6 is the frame pointer.
        adda.l   #-4,a7             Create space for locals.
```

Listing 4.5: LINK Effective Code

With a6 as the frame pointer, the code can access local variables using a negative offset from A6, and access the function parameters with a positive offset from A6. Any working registers pushed onto the stack will go below the space required for the local variables used in the function. At this point, the stack looks like Figure 4.1

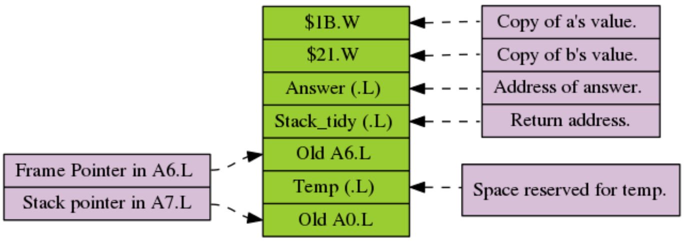


Figure 4.1: The stack structure

After setting the *temp* local variable to zero, the calculation is done and the result stored in the long word pointed to by *result* which is the address of the variable *Answer*, and the stack is tidied by popping A0 and then by unwinding the stack frame previously allocated using the `UNLK` instruction, which *effectively*, does this:

```
        move.l   a6,a7                Set a7 back again.
        move.l   (a7)+,a6             Retrieve previous a6.
```

Listing 4.6: UNLK Effective Code

And now, A7 points once again, at the return address in main, where execution will continue. The local variable *temp* is no more, it has ceased to be, it has shuffled off its mortal coil and gone to meet its maker, etc.[1]

[1] Monty Python's Dead Parrot sketch.

5. LibGen News

And it's good news, of a sort!

As my wife was away for the weekend recently, I took the opportunity to spend some time going through the problems I have been having with LibGen.

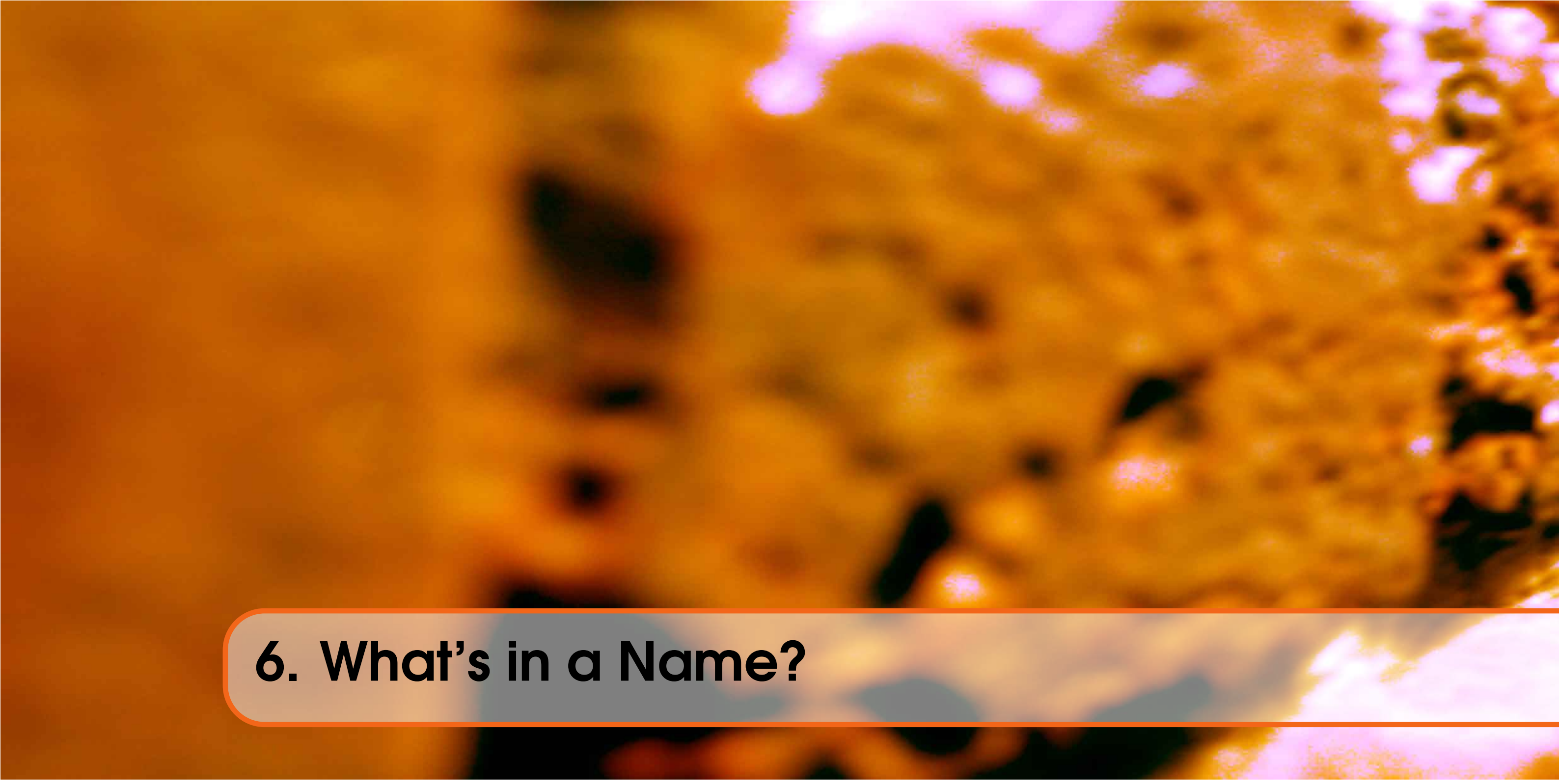
If you remember, I had managed to reach a stage where the program would assemble and execute, but on exiting, QPC would be trashed in as much as the cursor passed *behind* the window for QPC and therefore I was unable to get any further work done within QPC and I had to kill it.

This problem could be reproduced at will, and was apparent even if all I did was `ex LibGen_exe` and then, when it was running, pressed ESC to escape. QPC was hosed at this point.

It turned out that some modifications I had made to the window definition, in order to allow me the ability to create a new application sub-window menu dynamically, had caused the problem. One problem that I did notice was that I had set the pointer to the *menu items* status bytes to be zero, which meant that it used the status bytes for the *main window's* loose items instead.

There were probably other errors as well, but in the end, I recreated the Window using SETW as per the original article in QL Today, and everything is fine again.

So, the good news is, I've got a working starting point for the rest of the development. The bad news? Time is never on my side!



6. What's in a Name?

A thread on the QL Forum, entitled *Command Line Parameters* mentioned at one point, the ability to get a parameter as a name rather than a string. Now in all my years of Assembly programming, writing DJToolkit etc, I've never really bothered with names. The following listing is a small example of how to copy a single name parameter as passed to a procedure or function written in Assembly.

It does not do anything useful, other than take the name passed, run some checks on it, then if valid, copies it to a buffer and prints it to SuperBasic channel #1 which is *assumed to be open*.

6.1 The Code

```
;==============================================================
; A test routine to fetch a name from the supplied parameters to
; a PROCedure in this case, which keeps things simple. The name
; in question is copied to a buffer, then printed to channel #1.
; That is all.
;==============================================================
; USAGE:
;
; GetName #1, something_not_in_quotes
;
; GetName fred_txt
;==============================================================


start       lea         define,a1       Procedure definition block.
            move.w      bp_init,a2      Initialise Procs/FNs.
            jmp         (a2)            Do it, exit to SuperBasic.

define      dc.w        1               One Procedure.
```

```
            dc.w        getName-*           Starting address offset.
            dc.b        7,'GetName'         Name of procedure.
            dc.w        0                   End of Procedures.

            dc.w        0                   There are zero Functions.
            dc.w        0                   The end of those too.

buffer      ds.w        1+512               Word count and 1024 bytes.
```

Listing 6.1: GetName - Definition Block

We start the code with the standard new procedure and/or function definition block. Following this is a buffer of 1024 bytes and an extra word for the usual QDOSMSQ string length. You will notice I’ve used `ds.w` instead of `ds.b` to ensure that the buffer starts on a word boundary.

```
;=============================================================
; A name table entry is 8 bytes, as follows:
;
; Offset  Size  Description
;    0    word  Type
;    2    word  Index of name in name list, or -1 (expression.)
;    4    long  Offset into variables area for value of this
;               name, or SuperBasic line number, (SB Functions &
;               Procs) or Absolute address in memory (for MC
;               Functions/Procs).
;=============================================================
; A name list entry is 'n' bytes, as follows:
;
; Size  Description
; byte  Length of this name. NOT word aligned.
; bytes Bytes of name.
;=============================================================
```

Listing 6.2: GetName - Name Table & Name List Definition

The comment above simply reminds us (me!) of what a name table entry looks like. Each entry is 8 bytes and on entry to a procedure or function, A3 and A5 point, relative to A6, at the first and last of the supplied parameters.

In the parameter list, the byte at offset 1 holds details of the separators used in the parameter list. This is not used in the main name table though.

```
;=============================================================
; A name list entry is 'n' bytes, as follows:
;
; Size  Description
; byte  Length of this name. NOT word aligned.
; bytes Bytes of name.
;=============================================================

err_bp      equ         -15                 Bad parameter error code.
bv_ntbas    equ         $18                 Offset to Name Table.
bv_nlbas    equ         $20                 Offset to Name List.
bv_chbas    equ         $30                 Offset to channel table.
```

Listing 6.3: GetName - Equates

Another comment reminds us of how each entry in the name list looks, and is followed by a few equates that will be used later.

Now we get to the meat of the code.

```
getName     tst.b       0(a3,a6.l)          Is the type a NULL?
            beq.s       nameFound           No, bale out, not a name.

bp_error    moveq       #err_bp,d0          Bad parameter
            rts                             We are out of here!
```

Listing 6.4: GetName - Checking Parameters

We begin by testing to see if the type byte of the first parameter passed is unset, which indicates a name. If it isn't a name, we bale out to SuperBasic with a bad parameter error.

```
nameFound   movea.l     bv_ntbas(a6),a0     Name Table start in A0.
            move.w      2(a3,a6.l),d0       Name list index number.
            lsl.w       #3,d0               Multiply by 8.
            adda.w      d0,a0               A0 = Name Table entry.
```

Listing 6.5: GetName - We Have a Name

Here we know we have a name, so we begin by getting the offset of the start of the name table into A0. From the passed parameter details, we extract the index number of this parameter's entry in the real name table (the parameter entries are *copies* and as each entry is 8 bytes, a quick shift three bits left will do the multiplication for us.

Adding D0 to A0 gets us the offset from A6 where we can find this name in the name table.

```
;=============================================================
; Now, from A0's position in the Name Table, access the Name
; List, relative to A6 of course.
;=============================================================

            move.w      2(a0,a6.l),d0       Offset into the Name List.
            ext.l       d0                  Make it long.
            add.l       bv_nlbas(a6),d0     D0 = Name List offset.

;=============================================================
; We now have the text of the name, in the name list, at the
; offset in D0.
;=============================================================
```

Listing 6.6: GetName - Find it in the Name List

In the name table, we pick up the offset into the name list for this name. The name list holds the actual characters of the name. As ever, everything is relative to A6.

```
            lea         buffer,a3           Destination buffer.
            moveq       #0,d1               Clear length WORD.
            move.b      0(a6,d0.l),d1       Get length BYTE.
            clr.b       0(a3)               Buffer size word top byte
;                                           must be zero.

copy_name   move.b      0(a6,d0.l),1(a3)    Copy one byte into buffer.
            addq.l      #1,a3               Next free space in buffer.
            addq.l      #1,d0               Next char in Name List.
            dbra        d1,copy_name        Copy size byte plus name
```

```
;                                       bytes.
            move.b      #linefeed,1(a3) And tag on a linefeed.
```

Listing 6.7: GetName - Copy Name to Buffer

A3 is set to the address of the destination buffer for the characters in this name and d1.w is cleared as we need a word sized counter. As the name list entry is byte sized, we get the length into D1's lower byte.

Normally, we would decrement D1.w before we start copying bytes, but in this case, we are copying the size byte from the name list, so we keep hold of the extra one byte in the counter to account for that.

The first byte in the buffer is cleared as the length word's high byte can never be anything but zero when copying from the name list.

The loop at `copy_name` copies first the size byte and then all the characters of the name into the buffer one by one. When we are done copying, the linefeed character is stored at the end of the name's bytes.

You will note, at this point, that the length word at the start of the buffer has no idea that the linefeed has been added. We are keeping it in the dark for now.

Looking at the above code, I should really have got rid of all those 1(An) offset instructions and started with a post increment of A3 or similar, but hey, the code works! I'll probably get a telling off from George though! ;-)

```
;==============================================================
; Now we have the text of the name in our buffer. Find channel
; #1 in the channel table. We shouldn't be off the end of the
; table, so NOT CHECKED.
; We assume #1 is open too, so that's NOT CHECKED for either.
;==============================================================
findChan    moveq       #40,d1            Offset to entry #1.
            move.l      bv_chbas(a6),a0   Channel table base offset.
            adda.l      d1,a0             Required entry for #1.
            move.l      0(a6,a0.l),a0     A0 is ID of channel #1.
```

Listing 6.8: GetName - Checking Channel #1

The code above deep dives into the SuperBasic channel table. It takes no account of where the end of the table might be, nor even if channel #1 is closed or not. It assumes much. Production code would never do such a thing!

Each entry is 40 bytes long, and the channels number from zero, so we need the *second* entry in the table.

A0 is set to the start of the channel table, D1 holds the offset to #1, and is added to A0. The first long word in each entry is the channel id as far as QDOSMSQ is concerned. What SuperBasic knows as #1 could be anything, but back in the old days, was something like $00010001. But never assume this to be the case now.

```
;==============================================================
; Print the text we read from the name list to channel #1.
;==============================================================
printName   move.w      UT_MTEXT,a2       Vector to print a string.
            lea         buffer,a1         The string to print.
            addi.w      #1,(a1)           Include the linefeed
            jmp         (a2)                 Print it, and exit
;                                            to SuperBasic.
```

Listing 6.9: GetName - Printing the Name

And finally, with A0 holding the QDOSMSQ channel id, we point A1 at the buffer start and add 1 to the word stored there to account for that linefeed we sneaked in earlier. With the buffer now ready to print, we jump into QDOSMSQ to print the text to #1 on the screen and never return. If there are any errors in the printing of the name, SuperBasic will handle it.

6.2 How to Run the Code

Type the above into your favourite editor and assemble it. Then simply `LRESPR` the assembled file and the new routine named `GetName` is available for use and abuse. To run it, type the following:

```
GetName This_has_no_quotes
```

This example will simply print what you passed, on screen, wherever channel #1 happens to be. Remember to run this in a SuperBasic or Sbasic that has at least channel #1 open. Other examples could be file names:

```
GetName flp1_boot
GetName win1_source_qltoday_LibGen
```

And if you try passing a number or a string, then you should get a Bad Parameter error message.

6.3 What if There Are More Parameters?

The code example assumes only one parameter will be passed, but makes no checks. In real code, you might be expecting a number of parameters so you would check the numbers passed and their types before fetching them one by one (for the names) and then getting the others in groups as per normal.

You don't need to clean the values for names off the stack as they are never on it. You will, for the strings, integers etc. Not so much in procedures, but most definitely in functions.

7. QL Assembly - Comments by George Gwilt

The following is a list of observations and comments from George, on the first version of the QL_Assembly.pdf eBook, which was made available for download just before Christmas. Since then, it has been updated to include the following.

Here are some notes on your Assembly Language Programming Series.

1. The definition of `LEA` on page 37 should state that the effective address put into the address register is a long word. The official definition by Motorola states that the size is long.
   [ND] *Fixed.*
2. The `PEA` instruction is defined on page 39. As for `LEA` the size for `PEA` is long. This should be made clear.
   [ND] *Fixed.*
3. On page 39 you ask what use `PEA` is, when `LEA` could be used instead. There are three answers.
   (a) Using `LEA` requires the use of a register, such as A1, whereas `PEA` does not. It also needs one more instruction.
   (b) `PEA` allows you to choose between several subroutines but return to the same address form each. An example occurs in GWASS:

```
        PEA     INS_FP4     the return address
        BEQ     FP_XD
        BRA     FP_XS
```

Listing 7.1: PEA Example from Gwass Assembler

   (c) `PEA` can be used to put a number on the stack. EG

```
        PEA     4           puts 4 on the stack.
```

Listing 7.2: PEA Stacking a Literal Value

[ND] *I did cover these in the book, at least the part about needing two instructions, and a register.*

4. On page 40 the first line is wrong (as you can easily see!).
   [ND] *Yes indeed I can! Oops.*
5. On the same page you deal with `LINK`, suggesting that it is probably most used by compilers. The official Motorola User's Manual says that `LINK` and `UNLK` can be used to maintain a linked list of local variables and parameter areas on the stack for nested subroutine calls.
   As it happens I use `LINK/UNLK` in GWASS as part of the assembly of macros. Each area allocated by `LINK` is used to store the macro parameters. Since the number of these can vary from macro to macro, I need to use `LINK` with a variety of displacement values.
   Moreover, since macros can contain calls to other macros, the set of LINK/UNLK instructions can indeed be nested.
   In order to allow a variety of displacements I produce a table of pointers to the different `LINK` instructions needed. This, of course, is done by means of a macro.
   One problem with the use of nested `LINK`s is that each time you use a further one the available stack space becomes smaller. To avoid trouble I check for each new `LINK` that there will indeed be enough space for it.
   [ND] *See elsewhere in this issue for a few examples of PEA, LINK and UNLK.*
6. Section 6.4 deals with exceptions. The descriptions of the stack frame at the bottom of page 48 and the top of page 49 are upside down. I think this is copied (wrongly) from Pennel's QDOS Companion page 91. Also, the description on page 49 is an atypical exception stack frame and applies only to the 68000/8 Bus or Address Error.
   [ND] *It was actually copied from the official Motorola 68000 Programmer's Reference Manual, 4th Edition page 39. On that page there is a diagram of the MC68000 and MC68008 Group 1 and Group 2 Exception Stack Frame which shows the SSP pointing at the Status Register at the low address of the stack frame, then the PC high word and PC low word are next, going up in memory.*
   I wonder if the Motorola book is wrong?
   The final line on page 48 explains that the diagram on page 49 is indeed for a BUS ERROR, ADDRESS EROR or a RESET exception and that those three differ from all the others.]
7. Section 6.5 deals with a redirection of some of the traps and exception vectors. These range from address error to trap #15. You then show how to program each exception handler. I would very much suggest that this is definitely something to avoid. The main reason for `MT.TRAPV` probably is to allow the user to alter only one or two of the handlers, in particular the traps numbered 5 to 15, which are not used by QDOS.
   [ND] *Fair point. The example did show redefining all the available vectors, which could be handy, in a debugger/monitor perhaps. I agree that redefining one or two might be more common.*
8. A minor point in 7.2 on page 54 is that I would use

   ```
       jmp (a2)
   ```

   Listing 7.3: Saving an RTS Instruction

   instead of

   ```
       jsr (a2)
       rts
   ```

   Listing 7.4: Wasting an RTS Instruction

   [ND] *Yes, I have a habit of doing that.*
9. You can operate doubly linked lists, described on page 118 by using only one pointer instead of two. Replace the two addresses, next (A say) and prior (B say) by their XOR combination (C say).

Thus

$$C = A \operatorname{xor} B$$

so that

$$B = A \operatorname{xor} C$$

and

$$A = B \operatorname{xor} C$$

[ND] *This is quite neat, and I have seen it used before, a long time back. I suspect back then there was a need to save every possibly byte at the expense of having to use a couple more instructions to extract the data required - but I am rather fond of the XOR operation, I have to say.*

To illustrate how such a doubly linked list can be operated I have produced a small PE program. This has loose items A, D, H and W.

- A adds an item (to the start of the list).
- D deletes an item from the list.
- H prints the number of items in the list.
- W prints, in hex, the address of an item.

Since this program is designed to show how to perform these operations not as a real working program with a real list, the list is constrained to consist of items which are simply a digit between 0 and 9 inclusive.

The minimum initial information you need is the address of the first item, stored at fadd(A6), and the address of the last item, stored at ladd(A6).

These are made zero when the program starts so that initially there is no list.

The program is given below.

```
; LIST a_asm


            bra.s       start
            dc.l        0
            dc.w        $4afb
fname       dc.w        fname_e-fname-2
            dc.b        "LIST v1.01"
fname_e   ds.b          0
            ds.w        0

            in          win1_ass_pe_keys_pe
            in          win1_ass_pe_qdos_pt
            in          win1_ass_pe_keys_wwork
            in          win1_ass_pe_keys_wstatus
            in          win1_ass_pe_keys_wman
            in          win1_ass_pe_keys_wdef
            in          win1_lib_hed1

            rsset       0
id          rs.l        1
wmvec       rs.l        1
slimit      rs.l        1
fadd        rs.l        1
ladd        rs.l        1
num         rs.l        1                       long int for conversion
buf         rs.l        2                       ASCII hex of num
*
```

```
start     lea       (a6,a4.l),a6           dataspace
          clr.l     fadd(a6)               mark ..
          clr.l     ladd(a6)               .. no list
          bsr.s     ope                    open a con channel ..
          move.l    a0,id(a6)              .. keep the ID
          moveq     #iop_pinf,d0
          moveq     #-1,d3
          trap      #3
          tst.l     d0                     ptr_gen present? ..
          bne       sui        ---->       .. no
          move.l    a1,wmvec(a6)           keep WM vector ..
          beq       sui        ---->       .. wasn't there!
          movea.l   a1,a2                  set WM vector in A2
          lea       slimit(a6),a1
          moveq     #0,d2                  this must be zero
          moveq     #iop_flim,d0           max size of window ..
          trap      #3
          subi.l    #$C0008,(a1)           .. less 12, 8
          lea       wd0,a3                 window definition addr
          move.l    #ww0_0,d1  size of     working definition ..
          bsr       getsp                  .. sets ALCHP'd addr ..
          movea.l   a0,a4                  .. to A0 and to A4

; We need to set the status area to zeros
; and the loose items to "available" (zero)

          lea       wst0,a1                Status ..
          movea.l   a1,a0                  .. area ..
          moveq     #wst0_e-wst0-1,d1      bytes to clear - 1
st1       clr.b     (a0)+
          dbf       d1,st1
          movea.l   id(a6),a0              Replace the channel ID
          move.l    wd_xmin+wd_rbase(a3),d1      minimum size
          andi.l    #$FFF0FFF,d1           Lop off scaling factors
          jsr       wm_setup(a2)           Set up working defn
          moveq     #-1,d1                 Set the window ..
          jsr       wm_prpos(a2)           .. where the pointer is
          jsr       wm_wdraw(a2)           Draw the contents
wrpt      jsr       wm_rptr(a2)            Read the pointer

          beq.s     no_err                 Since D0 is zero then ..
;                                          .. D4 is non zero
          bra       sui        ---->       D0 is non zero


*

con       dc.w      3
          dc.b      'con'

ope       lea       con,a0                 To open "con" ..
          moveq     #-1,d1                 .. for this job
          moveq     #0,d3
          moveq     #io_open,d0
          trap      #2
          rts
```

```

; We come here if we exit from wm_rptr without an error
; This means that D4 is non-zero which in turn means either that
; there was a window event (eg CTRL/F4) or that a loose item
; action routine has set a non-zero value in D4. If there was a
; window event (and no loose item) the appropriate bit will have
;  been set in the event vector in the status area.

; If a loose item has a select key equal to that for an event,
; the event will not be detected by WM_RPTR since the loose
; item's action routine will have been called instead. The loose
; item's action routine can then set the event bit in the event
; vector and force an exit from WM_RPTR by setting the event
; number in D4. In that case the following code would be used.
; On the other hand the loose item's action routine could
; process the event internally without exiting from WM_RPTR.

no_err      movea.l     (a4),a1        status area
            btst        #p          t__can,wsp_weve(a1)
            bne         sui            Exit

            btst        #pt__move,wsp_weve(a1)
            beq.s       wrpt
            bsr         move
            bra.s       wrpt


; Loose item action routines

; MOVE

afun0_0     bsr         move
af1         move.w      wwl_item(a3),d1                     item number
            move.b      #wsi_mkav,ws_litem(a1,d1.w)         ask for redraw
            moveq       #-1,d3                              selective draw
            jsr         wm_ldraw(a2)
            clr.b       ws_litem(a1,d1.w)                   available
            moveq       #0,d4
            moveq       #0,d0
            rts

; EXIT

afun0_3 moveq           #0,d0
            moveq       #pt__can,d4                         ESC
            bset        #pt__can,wsp_weve(a1)
            rts

; A - Add an item to the list

afun0_1 move.l          a1,-(sp)
            bsr         dwin
            bsr         cls                     clear window
            lea         pt_1,a5                 text
            bsr         mtext
            moveq       #-1,d3
```

```
          moveq     #0,d7
          moveq     #io_fbyte,d0            item in D1.B
          bsr       tp3
          move.b    d1,d7
          subi.l    #'0',d7                 0 to 9 (we hope)
          bmi       af1_er     ---->        (te6)
          cmpi.b    #9,d7
          bgt       af1_er     ---->
          bsr       add_it
          beq       af1_2                   OK        (te5)
          bmi       af1_3                   duplicate (te4)
          lea       te2,a5                  list full (te2)
          bra       af1_4
af1_er    lea       te6,a5
          bra       af1_4
af1_2     lea       te5,a5
          bra       af1_4
af1_3     lea       te4,a5
af1_4     bsr       mtext
          movea.l   (sp)+,a1
          bra       af1


; W - Where is the item?

afun0_2 move.l      a1,-(sp)
          bsr       dwin
          bsr       cls                     clear window
          lea       pt_2,a5                 text
          bsr       mtext
          moveq     #-1,d3
          moveq     #0,d7
          moveq     #io_fbyte,d0            item in D1.B
          bsr       tp3
          move.b    d1,d7
          subi.l    #'0',d7                 0 to 9 (we hope)
          bmi       af1_er     ---->        (te6)
          cmpi.b    #9,d7
          bgt       af1_er     ---->
          bsr       there
          beq       af5_3                   Not There
          bpl       af2_1                   OK
          lea       te1,a5
          bra       af1_4
af2_1     move.l    d0,num(a6)              for printingn num(A6)
          movem.l   a0/a2-3,-(sp)           keep regs
          lea       num,a1                  arithmetic buffer
          lea       buf,a0                  space for answer
          movea.w   cn_itohl,a2
          jsr       (a2)
          movem.l   (sp)+,a0/a2-3           replace regs
          lea       buf(a6),a1              for printing
          moveq     #8,d2                   to print 8 bytes
          moveq     #io_sstrg,d0
          bsr       tp3
          movea.l   (sp)+,a1                reset A1
          bra       af1                 return

; H - How many in the list?

afun0_4 move.l      a1,-(sp)
          bsr       dwin
          bsr       cls
          bsr       howmany             number -> A4
          move.w    d4,d1
          movem.l   a2-3,-(sp)
          movea.w   ut_mint,a2
          jsr       (a2)
          movem.l   (sp)+,a2-3
          movea.l   (sp)+,a1
          bra       af1

; D - Delete an item from the list

afun0_5 move.l      a1,-(sp)
          bsr       dwin
          bsr       cls                 clear window
          lea       pt_5,a5   text
          bsr       mtext
          moveq     #-1,d3
          moveq     #0,d7
          moveq     #io_fbyte,d0        item in D1.B
          bsr       tp3
          move.b    d1,d7
          subi.l    #'0',d7             0 to 9 (we hope)
          bmi       af1_er    ---->     (te6)
          cmpi.b    #9,d7
          bgt       af1_er    ---->
          bsr       drop_it
          bne       af5_1               OK
af5_3     lea       te3,a5              'not there'
          bra       af5_2
af5_1     lea       te7,a5              'value dropped'
af5_2     bsr       mtext
          movea.l   (sp)+,a1
          bra       af1

          hed1      <'A'>,t1
          hed1      <'W'>,t2
          hed1      <'H'>,t3
          hed1      <'D'>,t4

dwin      move.l    a1,-(sp)
          moveq     #0,d1
          moveq     #7,d2
          jsr       wm_swinf(a2)
          movea.l   (sp)+,a1
          rts

cls       moveq     #-1,d3
          moveq     #sd_clear,d0
tp3       trap      #3
```

```
          rts

; mtext prints the string @A5

mtxt_reg  reg       d1-2/a1-3
mtext     movem.l   mtxt_reg,-(sp)
          movea.w   ut_mtext,a2
          movea.l   a5,a1
          jsr       (a2)
          movem.l   (sp)+,mtxt_reg
          rts

; Adds item with value D7.L
; D0 = 0  if OK : + if full : - if already there
ai_reg    reg       d0-1/d4/a0-1
add_it    movem.l   ai_reg,-(sp)
          bsr       howmany
          cmpi.w    #9,d4
          ble       ai5                 OK
          move.w    #1,d0
          bra       ai2                 Full
ai5
          bsr       there
          ble       ai3                 not already there
          moveq     #-1,d0              mark alredy there
          bra       ai2
ai3       moveq     #8,d1
          bsr       getsp
          move.l    d7,4(a0)            Set item value
          tst.l     fadd(a6)
          bne       ai1
          move.l    a0,fadd(a6)
          move.l    a0,ladd(a6)
          clr.l     (a0)
          bra       ai4
ai1       movea.l   fadd(a6),a1
          move.l    a1,(a0)
          move.l    (a1),d0
          move.l    a0,d1
          eor.l     d1,d0
          move.l    d0,(a1)             update pointers
          move.l    a0,fadd(a6)         new start address
ai4       moveq     #0,d0               mark OK
ai2       movem.l   (sp)+,ai_reg
          rts

; To delete the item with value in D7.L
; First find the item then delete it
; On exit D0.L = 0 NOT THERE
;             = 1 Done OK

di_reg    reg       d0-1/d4-6/a0/a2/a4
drop_it   movem.l   di_reg,-(sp)
          bsr       there
          beq       di6       not there
          movea.l   d0,a4
          move.l    (a4),d1
          eor.l     d6,d1     next address



; D6.L = previous address
; A4.L and D0.L = address to be deleted
; D1.l = next address

          bsr       rechp     return item space to the heap
          tst.l     d6
          bne       di3       there is a previous address
          tst.l     d1
          bne       di4       there is a next address

; here the list is only the item to be deleted!!

          clr.l     fadd(a6)
          clr.l     ladd(a6)
          bra       di8

; next but no previous

di4       move.l    d1,fadd(a6)         new 1st address
          movea.l   d1,a0
di7       move.l    a4,d0
          eor.l     d0,(a0)
di8       moveq     #1,d0               mark OK
di6       movem.l   (sp)+,di_reg
          rts

di3       tst.l     d1
          bne       di5                 both previous and
;                                       next addresses

; previous but no next

          move.l    d6,ladd(a6)         new last address
          movea.l   d6,a0
          bra       di7

; Both before (B) and after (A) the current (C)

di5        movea.l  d1,a0
           move.l   (a0),d3             AC
           movea.l  d6,a0
           move.l   (a0),d4             BC
           move.l   (a4),d5             CC

           move.l   a4,d0               C -> D0

           eor.l    d0,d4
           eor.l    d1,d4               New BC

           eor.l    d0,d3
           eor.l    d6,d3               New AC
```

```

                movea.l   d1,a0
                move.l    d3,(a0)              set New AC

                movea.l   d6,a0
                move.l    d4,(a0)              set New BC

                bra       di8




; Returns the number of items in the list in D4.W
; Uses no other registers
hm_reg          reg       d1-3/a0
howmany         movem.l   hm_reg,-(sp)
                clr.w     d4
                move.l    fadd(a6),d1          1st address
                beq       hm1                  none!!!
                clr.l     d2
                addq.w    #1,d4                advance count
hm2             movea.l   d1,a0
                move.l    (a0),d3              pointer
                eor.l     d2,d3                next address
                beq       hm1                  finished
                move.l    d1,d2                new previous
                move.l    d3,d1                new current
                addq.w    #1,d4                advance count
                bra       hm2
hm1             movem.l   (sp)+,hm_reg
                tst.w     d4
                rts                            number in D4.W

; There returns in D0.L the address of the item with value in D7
; D7.L and in D6.L the previous address.
; If not found D0.L = 0,
; if list empty D0.L = -1
; Uses no other registers

th_reg          reg       d4/a0/a2
there           movem.l   th_reg,-(sp)
                clr.l     d6                   previous address
                move.l    fadd(a6),d0          1st address
                beq       th4                  List Empty
                bra       th1
th2             movea.l   d0,a2
                move.l    d6,d4
                move.l    d0,d6
                move.l    (a2),d0              pointer
                eor.l     d4,d0                next address
                beq       th3                  not found
th1             movea.l   d0,a0
                cmp.l     4(a0),d7             found? . .
                bne       th2                  . . no
th3             movem.l   (sp)+,th_reg
                tst.l     d0                   zero = not found:
```

```
;                                           + = found:
;                                           - = empty
            rts

th4         moveq       #-1,d0              mark 'empty'
            bra         th3

; program list

pr_lst      dc.w        afun0_1-pr_lst
            dc.w        afun0_2-pr_lst
            dc.w        afun0_4-pr_lst
            dc.w        afun0_5-pr_lst

; string list

pt_lst      dc.w        pt_1-pt_lst
            dc.w        pt_2-pt_lst
            dc.w        pt_4-pt_lst
            dc.w        pt_5-pt_lst

            hed1        <'Give value to add '>,pt_1
            hed1        <'Give value to find '>,pt_2
            hed1        <'Size is '>,pt_4
            hed1        <'Give value to delete '>,pt_5

; messages

            hed1        <'List Empty'>,te1
            hed1        <'List Full'>,te2
            hed1        <'Not There'>,te3
            hed1        <'Duplicate Item'>,te4
            hed1        <'Value Added'>,te5
            hed1        <'Out of Range'>,te6
            hed1        <'Value Dropped'>,te7

            in          win1_ass_pe_listw_asm

            in          win1_ass_pe_peas_sym_lst
            lib         win1_ass_pe_peas_bin


            in          win1_ass_pe_csprc_sym_lst

            lib         win1_ass_pe_csprc_bin
```

Listing 7.5: George's Linked List Example Program

Thanks George. I appreciate your taking the time to go over some stuff I wrote many years ago, and bringing these "problems" to my attention.